Concurrent Hash Tables: Fast and General(?)!

TOBIAS MAIER and PETER SANDERS, Karlsruhe Institute of Technology, Germany
ROMAN DEMENTIEV, Intel Deutschland GmbH, Germany

Concurrent hash tables are one of the most important concurrent data structures, which are used in numerous applications. For some applications, it is common that hash table accesses dominate the execution time. To efficiently solve these problems in parallel, we need implementations that achieve speedups in highly concurrent scenarios. Unfortunately, currently available concurrent hashing libraries are far away from this requirement, in particular, when adaptively sized tables are necessary or contention on some elements occurs.

Our starting point for better performing data structures is a fast and simple lock-free concurrent hash table based on linear probing that is, however, limited to word-sized key-value types and does not support dynamic size adaptation. We explain how to lift these limitations in a provably scalable way and demonstrate that dynamic growing has a performance overhead comparable to the same generalization in sequential hash tables.

We perform extensive experiments comparing the performance of our implementations with six of the most widely used concurrent hash tables. Ours are considerably faster than the best algorithms with similar restrictions and an order of magnitude faster than the best more general tables. In some extreme cases, the difference even approaches four orders of magnitude.

All our implementations discussed in this publication can be found on github [17].

CCS Concepts: • **Theory of computation → Sorting and searching**; **Shared memory algorithms**; *Bloom filters and hashing*; *Data structures and algorithms for data management*; • **Information systems →** *Data structures*; • **Software and its engineering →** *Data types and structures*; • **Computer systems organization →** Multicore architectures;

Additional Key Words and Phrases: Concurrency, dynamic data structures, experimental analysis, hash table, lock-freedom, transactional memory

ACM Reference format:
Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)!. *ACM Trans. Parallel Comput.* 5, 4, Article 16 (February 2019), 32 pages.
https://doi.org/10.1145/3309206

1 INTRODUCTION

A hash table is a dynamic data structure that stores a set of elements that are accessible by their key. It supports insertion, deletion, find, and update in constant expected time. In a concurrent hash table, multiple threads have access to the same table. This allows threads to share information in a flexible and efficient way. Therefore, concurrent hash tables are one of the most important

Authors' addresses: T. Maier and P. Sanders, Karlsruhe Institute of Technology, Kaiserstraße 12, Karlsruhe, 76131, Germany; emails: {t.maier, sanders}@kit.edu; R. Dementiev, Intel Deutschland GmbH, Am Campeon 10-12, Neubiberg, 85579, Germany; email: roman.dementiev@intel.com.

2329-4949/2019/02-ART16 $15.00
https://doi.org/10.1145/3309206

concurrent data structures. See Section 4 for a more detailed discussion of concurrent hash table functionality.

To show the ubiquity of hash tables, we give a short list of example applications: A very simple use case is storing sparse sets of precomputed solutions (e.g., References [3, 30]). A more complicated scenario for hash tables is at the heart of many data base–like applications (for example, in the case of aggregations `SELECT FROM...COUNT...GROUP BY` x [26]). Such a query selects rows from one or several relations and counts for every key x how many rows have been found (similar queries work with `SUM`, `MIN`, or `MAX`). Hashing can also be used for a data-base join [5]. Another group of examples is the exploration of a large combinatorial search space where a hash table is used to remember the already explored elements (e.g., in dynamic programming [39], itemset mining [31], a chess program, or when exploring an implicitly defined graph in model checking [40]). Similarly, a hash table can maintain a set of cached objects to save I/Os [27]. Further examples are duplicate removal, storing the edge set of a sparse graph to support edge queries [24], maintaining the set of nonempty cells in a grid-data structure used in geometry processing (e.g., [7]), or maintaining the children in tree data structures such as van Emde-Boas search trees [6] or suffix trees [22].

Many of these applications have in common that—even in the sequential version of the program—hash table accesses constitute a significant fraction of the running time. Thus, it is essential to have highly scalable concurrent hash tables that actually deliver significant speedups to parallelize these applications. Unfortunately, currently available general purpose concurrent hash tables do not offer the needed scalability (see Section 8 for concrete numbers). However, it seems to be folklore that a lock-free linear probing hash table can be constructed using atomic *compare and swap* (CAS) of key-value-pairs. Find-operations can even proceed naively and without any write operations. This implementation we call folkloreHT [39] has several restrictions that keep it from becoming the standard for parallel hash tables: keys and values have to be machine words to enable the necessary CAS operation (two-word CAS), the table cannot grow over its preallocated size, and it does not support true deletion (in a way that reclaims memory). In Section 4, we explain our own implementation of (folkloreHT) in detail, after elaborating on some related work, and introducing the necessary notation (in Sections 2 and 3, respectively). To see the potential large performance differences, consider a situation with mostly read only access to the table and heavy contention for a small number of elements that are accessed again and again by all threads. folkloreHT actually profits from this situation, because the contended elements are likely to be replicated into local caches. However, any implementation that needs locks or CAS instructions for find-operations, will become much slower than the sequential code on current machines. The purpose of our article is to document and explain performance differences, and, more importantly, to explore to what extent we can make folkloreHT more general with an acceptable deterioration in performance.

These generalizations are discussed in Section 5. We explain how to grow (and shrink) such a table, and how to support deletions and more general data types. In Section 6, we explain how hardware transactional memory synchronization can be used to speed up insertions and updates and how it may help to handle more general data types.

After describing implementation details in Section 7, Section 8 experimentally compares our hash tables with six of the most widely used concurrent hash tables for microbenchmarks including insertion, finding, and aggregating data. We look at both uniformly distributed and skewed input distributions. Section 9 summarizes the results and discusses possible lines of future research.

2 RELATED WORK

This publication follows up on our previous findings about generalizing fast concurrent hash tables [19, 20]. In addition to describing how to generalize a fast linear probing hash table, we offer an extensive experimental analysis comparing many concurrent hash tables from several libraries.

There has been extensive previous work on concurrent hashing. The widely used textbook "The Art of Multiprocessor Programming" [11] devotes an entire chapter to concurrent hashing and gives an overview over previous work. However, it seems to us that a lot of previous work focuses more on concepts and correctness but surprisingly little on scalability. For example, most of the discussed growing mechanisms in Reference [11] assume that the size of the hash table is known exactly without a discussion that this introduces a performance bottleneck limiting the speedup to a constant. In practice, the actual migration is often done sequentially.

Stivala et al. [39] describe a bounded concurrent linear probing hash table specialized for dynamic programming that only supports insert and find. Their insert operation starts from scratch when the CAS fails, which seems suboptimal in the presence of contention. An interesting point is that they need only word size CAS instructions at the price of reserving a special empty value. This technique could also be adapted to port our code to machines without 128-bit CAS. Kim and Kim [13] compare this table with a cache-optimized lock-free implementation of hashing with chaining and with hopscotch hashing [12]. The experiments use only uniformly distributed keys, i.e., there is little contention. Both linear probing and hashing with chaining perform well in that case. The evaluation of find-performance is a bit inconclusive: Chaining wins but uses more space than linear probing. Moreover, it is not specified whether this is for successful (search for keys of inserted elements) or mostly unsuccessful (search for not inserted keys) accesses. We suspect that varying these parameters could reverse the result.

Gao et al. [9] present a theoretical dynamic linear probing hash table that is lock-free. The main contribution is a formal correctness proof. Not all details of the algorithm or even an implementation is given. There is also no analysis of the complexity of the growing procedure.

Shun and Blelloch [37] propose *phase concurrent hash tables*, which are allowed to use only a single operation within a globally synchronized phase. They show how phase concurrency helps to implement some operations more efficiently and even deterministically in a linear probing context. For example, deletions can adapt the approach from Reference [14] and rearrange elements. This is not possible in a general hash table, since this might cause find-operations to report false negatives. They also outline an elegant growing mechanism albeit without implementing it and without filling in all the details like how to initialize newly allocated tables. They propose to trigger a growing operation when any operation has to scan more than $k \log n$ elements where k is a tuning parameter. This approach is tempting, since it is somewhat faster than the approximate size estimator we use. We actually tried that but found that this trigger has a very high variance—sometimes it triggers late making operations rather slow, sometimes it triggers early wasting a lot of space. We also have theoretical concerns, since the bound $k \log n$ on the length of the longest probe sequence implies strong assumptions on certain properties of the hash function. Shun and Blelloch make extensive experiments including applications from the problem based benchmark suite [38].

Li et al. [16] use the bucket cuckoo-hashing method by Dietzfelbinger and Weidling [8] and develop a concurrent implementation. They use fine grained per bucket locks which can sometimes be avoided using transactional memory, e.g., Intel Transactional Synchronization Extensions TSX (Intel TSX). To further reduce the number of acquired locks per insertions they use a BFS-based insertion algorithm to find a minimal displacement path to an empty bucket (in contrast to the commonly used random walk). As a result of their work, they implemented the small open source library libcuckoo, which we measure against (which does not use Intel TSX). This approach has the potential to achieve very good space efficiency. However, our measurements indicate that the performance penalty is high.

The practical importance of concurrent hash tables also leads to new and innovative implementations outside of the scientific community. A good example of this is the Junction library [34],

which was published by Preshing in the beginning of 2016, shortly after our initial publication [19].

3 PRELIMINARIES

We assume that each application thread has its own designated hardware thread or processing core and denote the number of these threads with p. A data structure is non-blocking if no blocked thread currently accessing this data structure can block an operation on the data structure by another thread. A data structure is lock-free if it is non-blocking and guarantees global progress, i.e., there must always be at least one thread finishing its operation in a finite number of steps.

Hash Tables store a set of $\langle Key, Value \rangle$ pairs (elements).[1] A hash function h maps each key to a cell of a table (an array). The number of elements in the hash table is denoted n and the number of operations is m. For the purpose of algorithm analysis, we assume that n and m are $\gg p^2$ – this allows us to simplify algorithm complexities by hiding $O(p)$ terms that are independent of n and m in the overall cost. Sequential hash tables support the insertion of elements, and finding, updating, or deleting an element with given key—all of this in constant expected time. Further operations compute n (*size*), build a table with a given number of initial elements, and iterate over all elements (*forall*).

Linear Probing is one of the most popular sequential hash table schemes used in practice. An element $\langle x, a \rangle$ is stored at the first free table entry following position $h(x)$ (wrapping around when the end of the table is reached). Linear probing is at the same time simple and efficient—if the table is not too full, a single cache line access will be enough most of the time. Deletion can be implemented by rearranging the elements locally [14] to avoid holes violating the invariant mentioned above. When the table becomes too full or too empty, the elements can be migrated to a larger or smaller table, respectively. The migration cost can be charged to insertions and deletions causing amortized constant overhead.

4 CONCURRENT HASH TABLE INTERFACE AND FOLKLORE IMPLEMENTATION

Although it seems quite clear what a hash table is and how this generalizes to concurrent hash tables, there is a surprising number of details to consider. Therefore, we will quickly go over some of our interface decisions and detail how this interface can be implemented in a simple, fast, lock-free concurrent linear probing hash table.

This hash table will have a bounded capacity c that has to be specified when the table is constructed. It is the basis for all other hash table variants presented in this publication. We call this table folkloreHT, because variations of it are used in many publications and it is not clear to us by whom it was first published.

The most important requirement for concurrent data structures is that they should be *linearizable*, i.e., it must be possible to order the hash table operations in some sequence—without reordering two operations of the same thread—so that executing them sequentially in that order yields the same results as the concurrent processing. For a hash table data structure, this basically means that all operations should be executed atomically some time between their invocation and their return. For example, a `find` should never return an inconsistent state, e.g., a half-updated data field that was never actually stored at the corresponding key.

Our variant of the folkloreHT ensures the atomicity of operations using two-word atomic CAS operations for all changes of the table. As long as the key and the value each only use one machine

[1] Much of what is said here can be generalized to the case when *Element*s are black boxes from which keys are extracted by an accessor function.

word, we can use two-word CAS opearations to atomically manipulate a stored key together with the corresponding value. There are other variants that avoid needing two-word compare and swap operations, but they often need a designated empty value (see [34]). Since, the corresponding machine instructions are widely available on modern hardware, using them should not be a problem. If the target architecture does not support the needed instructions, then the implementation could in theory be switched to use a variant of concurrent linear probing without two-word CAS. As it can easily be deduced by the context, we will usually omit the "two-word" prefix and use the abbreviation CAS for both single and double word CAS operations.

ALGORITHM 1: Pseudocode for the `insertOrUpdate` operation

Input: Key k, Data Element d, Update Function $up: Key \times Val \times Val \to Val$
default: `atomicUpdate(·) = CAS(`$current$, $up(k, current.data, d)$`)`
Output: Boolean `true` when a new key was inserted, `false` if an update occurred

```
1  i = h(k);
2  while true do
3      i = i % c;
4      current = table[i];
5      if current.key == empty_key then                    // Key is not present yet . . .
6          if table[i].CAS(current, ⟨k, d⟩) then return true;
7          else continue;                                          // retry at same position
8      else if current.key == k then                      // Same key already present . . .
9          if table[i].atomicUpdate(current, d, up) then return false;
10         else continue;                                          // retry at same position
11     i++;
```

Initialization. The constructor allocates an array of size c consisting of 128-bit aligned cells whose key is initialized to the empty values.

Modifications. We categorize all changes to the hash table's content into one of the following three functions that can be implemented similarly (excluding deletions).

`insert(` k, d `)`: Returns `true` if the insert succeeds (`false` if an element with the specific key is already present). Only one operation should succeed if multiple threads are inserting the same key at the same time. The implementation is similar to Algorithm 1, with lines 11–14 replaced by `return false`.

`update(` k, d, up `)`: Returns `false`, if there is no value stored at the specified key, otherwise this function atomically updates the stored value to $new = up(current, d)$. Notice that the resulting value can be dependent on both the current value and the input parameter d. The implementation is similar to Algorithm 1, with lines 6–9 replaced by `return false` and line 12 replaced by `return true`.

`insertOrUpdate(` k, d, up `)`: This operation updates the current value, if one is present, otherwise the given data element is inserted as the new value. The function returns `true`, if `insertOrUpdate` performed an `insert` (key was not present), and `false` if an `update` was executed.

We choose this interface for two main reasons. It allows applications to quickly differentiate between inserting and changing an element—this is especially useful, since the thread that first inserted a key can be identified uniquely. Additionally, it allows transparent, lock-free updates that can be more complex than just replacing the current value (think of CAS or Fetch-and-Add).

Using an update function in the update interface deserves some attention, because the interface changes the way both experienced and inexperienced users interact with the table. Most interfaces fall into one of two categories: They either return mutable references to table elements—forcing the user to implement atomic operations on the data type; or they offer an `update` function, which usually replaces the current value with a new one—making it very hard to implement atomic read modify write operations (`find` + `increment` + `overwrite` not necessarily atomic). We made sure that using our interface, experienced users can implement both atomic and non-atomic changes to our table. The correct behavior can be chosen according to the specific update function (line 8), e.g., overwrite (using single word store), increment (using fetch and add).

In Algorithm 1, we show the pseudocode of the `insertOrUpdate` function. The operation computes the hash value of the key (line 1) and proceeds to look for an element with the appropriate key (beginning at the corresponding position). If no element matching the key is found (when an empty space is encountered (line 5)), then the new element has to be inserted. This is done using a CAS operation (line 6). A failed swap can only be caused by another insertion into the same cell. In this case, we have to revisit the same cell (line 9), to check if the inserted element matches the current key. If a cell storing the same key is found, then it will be updated using the `atomicUpdate` function (line 11). This function is usually implemented by evaluating the passed update function (*up*) and using a CAS operation, to change the cell. In the case of multiple concurrent updates, at least one will be successful.

Lookup. For the performance of many hash table workloads lookup operations are even more important than table modifications. In folkloreHT, lookups can proceed without any memory changes.

`find(`k`)` returns the value that has been stored with the key k or $\perp$ if the key has not been stored. The following problem arises, because we use 64 bits each for the key and the value. This is necessary, because it allows storing a pointer as value. To implement a correct find operation one has to be aware that using modern hardware it is impossible to atomically read both the key and the value together[2] (together they have 128 bits). Therefore—even though any change to a cell is atomic—it is possible for a cell to be changed in between reading its key and its value, this is called a *torn read*. To argue the correctness of our `find` implementation, we have to make sure that torn reads cannot lead to any wrong behavior. There are two kinds of interesting torn reads: First an empty key is read while the searched key is inserted into the same cell. In this case, the element is not found (consistent, since it has not been fully inserted). Second, the element is updated between the key being read and the data being read. Since the data is read second, only the newer data is read. This is consistent with a finished update.

Deletions. `delete(`k`)` Returns `true` if an element with key k was present and is successfully removed. FolkloreHT does not support true deletions (deletions that reclaim previously used memory). Simply deleting elements from folkloreHT is not possible, because we have to make sure that future lookup operations can still find displaced elements. The same problem arises when rearranging elements.

The only way folkloreHT can support deletions is using dummy elements—called *tombstones*. Usually the key stored in a cell is replaced with `del_key`. Future operations will scan over deleted elements like over any other nonempty entry. This means that the cell cannot be used anymore. Using *tombstones* for handling deleted elements is usually not feasible, because the starting capacity has to be set dependent on the number of overall insertions (deletion does not free up any deleted

[2]The element is not read atomically, because x86 processor specifications do not guarantee atomicity for 128-bit reads.

cells). Even worse, *tombstones* will fill up the table and slow down find queries. In Section 5.4, we will show how our generalizations can be used to handle *tombstones* more efficiently.

No inconsistencies can arise from this kind of deletion. In particular, a concurrent find-operations with a torn read will return the element before the deletion, since the delete-operation will leave the value-slot a untouched. A concurrent insert $\langle x, b \rangle$ might read the key x before it is overwritten by the deletion and return *false*, because it concludes that an element with key x is already present. This is consistent with the outcome when the insertion is performed before the deletion in a linearization.

Forall. Many algorithms need to iterate over all elements within a hash table, e.g., outputting all unique elements after one round of duplicate detection. This problem can be solved easily, by iterating over all cells of the table and omitting empty cells. This approach is very cache efficient if the table is decently filled (constant ratio of filled cells). The use of dynamically sized hash tables enforces a constant fill ratios, because growing is only triggered once the table is decently filled. Forall operations can easily be done in parallel. For example by splitting the table into p equally sized parts or by splitting the table into more blocks and using an online scheduler to schedule these blocks to the different threads. The latter approach has the advantage that random element imbalances will likely be evened out.

Bulk Operations. Our experiments in Section 8.4 show that contentious updates (highly skewed key sequences) still have a large impact on running times (even when using atomic operations). The main reason for this is the number of cache faults produced by cache invalidation and the necessary sequenzialization through atomic operations. Bulk operations (see Section 5.5) might be a way to solve this problem. By sorting a number of hash table operations by their key, one can partition the table into sections where updates (or even insertions) can run in an embarrassingly parallel fashion.

Size. Keeping track of the number of contained elements deserves special notice here, because it turns out to be significantly harder in concurrent hash tables. In sequential hash tables, it is trivial to count the number of contained elements—using a single counter. This same method is possible in parallel tables using atomic fetch and add operations, but it introduces a massive amount of contention on one single counter creating a performance bottleneck. Because of this, we did not include a counting method in folkloreHT. In Section 5.2, we show how this can be alleviated using an approximate count.

5 GENERALIZATIONS AND EXTENSIONS

In this section, we detail how to adapt the concurrent hash table implementation—described in the previous section—to be universally applicable to all hash table workloads. Most of our efforts have gone into a scalable migration method that is used to move all elements stored in one table into another table. It turns out that a fast migration can solve most shortcomings of folkloreHT (especially deletions and adaptable size).

5.1 Storing Thread-Local Data

By itself, storing thread specific data connected to a hash table does not offer additional functionality, but it is necessary to efficiently implement some of our other extensions. Per-thread data can be used in many different ways, from counting the number of insertions to caching shared resources.

From a theoretical point of view, it is easy to store thread specific data. The additional space is usually only dependent on the number of threads ($O(p)$ additional space), since the stored data is often constant sized. Compared to the hash table this is usually negligible ($p \ll n$).

Storing thread specific data is challenging from a software design and performance perspective. Some of our competitors use a `register(·)` function that each thread has to call before accessing the table. This allocates some memory, that can be accessed using the global hash table object.

Our solution uses explicit handles. Each thread has to create a handle, before accessing the hash table. These handles can store thread specific data, since they are not shared between threads. This is not only in line with the RAII idiom (resource acquisition is initialization [25]), but it also protects our implementation from some performance pitfalls like unnecessary indirections and false sharing.[3] Moreover, the data can easily be deleted once the thread does not use the hash table anymore (delete the handle).

5.2 Approximating the Size

Keeping an exact count of the elements stored in the hash table can often lead to contention on one count variable. Therefore, we propose to support only an approximative size operation.

To keep an approximate count of all elements, each thread maintains a local counter of its successful insertions (using the method desribed in Section 5.1). Every $\Theta(p)$ such insertions this counter is atomically added to a global insertion counter I and then reset. Contention at I can be provably made small by randomizing the exact number of local insertions accepted before adding to the global counter, e.g., between 1 and p. I underestimates the size by at most $O(p^2)$. Since we assume the size to be $\gg p^2$ this still means a small relative error. By adding the maximal error, we also get an upper bound for the table size.

If deletions are also allowed, then we maintain a global counter D in a similar way. $S = I - D$ is then a good estimate of the total size as long as $S \gg p^2$. In this case, both global counts are updated every $\Theta(p)$ operations (independent of insert or delete), ensuring the same approximation quality as without deletions.

When a table is migrated for growing or shrinking (see Section 5.3.1), each migration thread locally counts the elements it moves. At the end of the migration, local counters are added to create the initial count for I (D is set to 0).

This method can also be extended to give an exact count—in absence of concurrent insertions/deletions. To do this, a list of all handles has to be stored at the global hash table object. A thread can now iterate over all handles computing the actual element size.

5.3 Table Migration

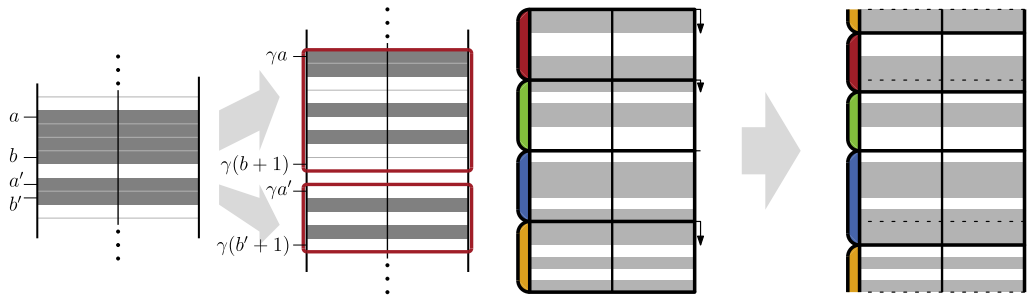
While Gao et al. [9] have shown that lock-free dynamic linear probing hash tables are possible, there is no result on their practical feasibility. Our focus is geared more toward engineering the fastest migration possible; therefore, we are fine with small amounts of locking, as long as it improves the overall performance.

The migration techniques we describe allow the table to work lock-free as long as no migration is triggered. Once a migration is triggered, it is not lock-free but it happens asynchronously and is hidden from any user of the hash table.

5.3.1 Eliminating Unnecessary Contention from the Migration. If the table size is not fixed, then it makes sense to assume that the hash function h yields a large pseudorandom integer, which is then mapped to a cell position in $0..c-1$ where c is the current capacity c.[4] We will discuss a way to do this by scaling. If h yields values in the global range $0..U-1$, then we map key x

[3] Significant slow down created by the cache coherency protocol due to multiple threads repeatedly changing distinct values within the same cache line.

[4] We use $x..y$ as a shorthand for $\{x, \ldots, y\}$ in this article.



(a) Two neighboring clusters and their non-overlapping target areas ($\gamma = 2$).

(b) Left: table split into even blocks. Right: resulting cluster distribution (moved implicit block borders).

Fig. 1. Cluster migration and work distribution.

to cell $h_c(x) := \lfloor h(x)\frac{c}{U} \rfloor$. Note that when both c and U are powers of two, the mapping can be implemented by a simple shift operation.

Growing. Now suppose that we want to migrate the table into a table that has at least the same size (growing factor $\gamma \geq 1$). Exploiting the properties of linear probing and our scaling function, there is a surprisingly simple way to migrate the elements from the old table to the new table in parallel, which results in exactly the same order a sequential algorithm would take and that completely avoids synchronization between threads.

Lemma 5.1. *Consider a range $a..b$ of nonempty cells in the old table with the property that the cells $a - 1 \bmod c$ and $b + 1 \bmod c$ are both empty—call such a range a* cluster *(see Figure 1(a)). When migrating a table, sequential migration will map the elements stored in that cluster into the range $\lfloor \gamma a \rfloor .. \lfloor \gamma(b + 1) \rfloor$ in the target table, regardless of the rest of the source array.*

Proof. Let x be an element stored in the cluster $a..b$ at position $p(x) = h_c(x) + d(x)$. Then, $h_c(x)$ has to be in the cluster $a..b$, because linear probing does not displace elements over empty cells ($h_c(x) = \lfloor h(x)\frac{c}{U} \rfloor \geq a$), and therefore, $h(x)\frac{c'}{U} \geq a\frac{c'}{c} \geq \gamma a$.

Similarly, $\lfloor h(x)\frac{c}{U} \rfloor \leq b$ implies that $h(x)\frac{c}{U} < b + 1$, and therefore, $h(x)\frac{c'}{U} < \gamma(b + 1)$. □

Therefore, two distinct clusters in the source table cannot overlap in the target table. We can exploit this lemma by assigning entire clusters to migrating threads, which can then process each cluster completely independently. Distributing clusters between threads can easily be achieved by first splitting the table into blocks (regardless of the tables contents), which we assign to threads for parallel migration. A thread assigned block $d..e$ will migrate those clusters that start within this range—implicitly moving the block borders to free cells as seen in Figure 1(b). Since the average cluster length is short and $c = \Omega(p^2)$, it is sufficient to deal out blocks of size $\Omega(p)$ using a single shared global variable and atomic fetch-and-add operations. Additionally, each thread is responsible for initializing all cells in its region of the target table. This is important, because sequentially initializing the hash table can quickly become infeasible. It is possible that a cluster covers a complete block. Using our simple scheduling method this would not be a problem, since this cluster fully belongs to the thread that works on the block it starts in. The covered block has no work to be done (clusters to migrate). The overall work balance will still be ensured by our simple work balancing mechanism (given that: $\#(\mathit{clusters}) \gg p$).

Note that waiting for the last thread at the end of the migration introduces some waiting (locking). Usually, this does not create significant work imbalance, since the block/cluster migration is very fast and clusters are expected to be short. But if the operating system would interrupt a

migrating thread, then all other threads would have to wait until this one thread is rescheduled and finishes its current block.

Shrinking. When elements are deleted, shrinking might be necessary to reused unused memory (see Section 5.4). Unfortunately, the nice structural Lemma 5.1 no longer applies. We can still parallelize the migration with little synchronization. Once more, we cut the source table into blocks that we assign to threads for migration. The scaling function maps each block $a..b$ in the source table to a block $a'..b'$ in the target table. We have to be careful with rounding issues so that the blocks in the target table are non-overlapping. We can then proceed in two phases. First, a migrating thread migrates those elements that move from $a..b$ to $a'..b'$. These migrations can be done in a sequential manner, since target blocks are disjoint. The majority of elements will fit into the target block. Then, after a barrier synchronization, all elements that did not fit into their respective target blocks are migrated using concurrent insertion, i.e., using atomic operations. This has negligible overhead, since elements like this only exist at the boundaries of blocks. The resulting allocation of elements in the target table will no longer be the same as for a sequential migration but as long as the data structure invariants of a linear probing hash table are fulfilled, this is not a problem.

5.3.2 Hiding the Migration from the Underlying Application. To make the concurrent hash table more general and easy to use, we would like to avoid all explicit synchronization. The growing (and shrinking) operations should be performed asynchronously when needed, without involvement of the underlying application. The migration is triggered once the table is filled to a factor $\geq \alpha$ (e.g., 50 %), this is estimated using the approximate count from Section 5.2, and checked whenever the global count is updated. When a growing operation is triggered, the capacity will be increased by a factor of $\gamma \geq 1$ (Usually $\gamma = 2$). The difficulty is ensuring that this operation is done in a transparent way without introducing any inconsistent behavior and without incurring undue overheads.

To hide the migration process from the user, we have to solve two problems. First, we have to find threads to grow the table (strategies u and p), and second, we have to ensure, that changing elements in the source table will not lead to any inconsistent states in the target table (possibly reverting changes made during the migration; strategies a and s). Each of these problems can be solved in multiple ways. We propose two strategies for each of them resulting in four different variants of the hash table (mix and match).

Recruiting User-Threads (u). A simple approach to dynamically assemble threads to growing the table, is to "enslave" threads that try to perform table accesses that would otherwise have to wait for the completion of the growing process anyway. This works well when the table is regularly accessed by all user-threads, but is inefficient in the worst case when most threads stop accessing the table at some point, e.g., waiting for the completion of a global computation phase at a barrier. The few threads still accessing the table at this point will need a lot of time for growing (up to $\Omega(n)$) while most threads are waiting for them. One could try to also enslave waiting threads but it looks difficult to do this in a sufficiently general and portable way.

Using a Dedicated Thread Pool (p). A provably efficient approach is to maintain a pool of p threads dedicated to growing the table. They are blocked until a growing operation is triggered. This is when they are awoken to collectively perform the migration in time $O(n/p)$ (assuming fair scheduling of migrating threads). Afterwards, they block again until the next migration is triggered. During a migration, application threads might have to sleep until the migration threads are finished. This will increase the CPU time of our migration threads making this method nearly as efficient as the enslavement variant. Using a reasonable computation model, one can show that using thread pools for migration increases the cost of each table access by at most a constant in a globally amortized sense (over the non-growing folkloreHT). We omit the relatively simple proof.

To remain fair in our experiments, we cannot use any additional processors (more than eventual competitors). Therefore, we create one growing thread per application thread accessing the hash table. Additionally, we pin each growing thread to the same core as the corresponding application thread.

Marking Moved Elements for Consistency (a—asynchronous). During the migration, it is important that no element can be changed in the old table after it has been copied to the new table. Otherwise, it would be hard to guarantee that changes are correctly applied to the new table. The easiest solution to this problem is to mark each cell before it is copied. Marking each cell can be done using a CAS operation to set a special marked bit, which is stored in the key. In practice this reduces the possible key space. If this reduction is a problem, then see Section 5.6 on how to circumvent it. To ensure that no copied cell can be changed, it suffices to ensure that no marked cell can be changed. This can easily be done by checking the bit before each writing operation and by using CAS operations for each update. This prohibits the use of fast atomic operations to change element values.

Whenever a thread t finds a marked element, it is clear that the current table is being migrated. Then t either helps with the migration, or blocks until the thread pool has finished migrating the table (depending on the migration strategy u or p). In principle, find operations could proceed without waiting for the migration. We chose to not enable this in our implementation.

After the migration, the old hash table has to be deallocated. Before deallocating an old table, we have to make sure that no thread is currently using it anymore. This problem can generally be solved by using reference counting. Instead of storing the table with a usual pointer, we use a reference counted pointer (e.g., `std::shared_ptr`) to ensure that the table is eventually freed.

The main disadvantage of counting pointers is that acquiring a counting pointer requires an atomic increment on a shared counter. Therefore, it is not feasible to acquire a counting pointer for each operation. Instead a copy of the shared pointer can be stored locally, together with the increasing version number of the corresponding hash table (using the method from Section 5.1). At the beginning of each operation, we can use the local version number to make sure that the local counting pointer still points to the newest table version. If this is not the case, then a new pointer will be acquired. This happens only once per version of the hash table. The old table will automatically be freed once every thread has updated its local pointer. Note that counting pointers cannot be exchanged in a lock-free manner increasing the cost of changing the current table (using a lock). This lock could be avoided by using a hazard pointer (we did not do this).

Preventing Concurrent Updates to Ensure Consistency (s—semi-synchronized). We propose a simple protocol inspired by read-copy-update protocols [23]. It uses p local flags $f_{1,..,p}$, and one global growing flag f_G to control that no thread is using the flag while it is being migrated. Whenever a thread t accesses the table, it sets its local busy flag f_t at the start of the operation and unsets it after the operation is completed (before returning the result of the operation). When a thread t_G triggers the growing operation it sets some global growing flag f_G using a CAS instruction. This global flag is inspected by each thread after setting its local flag (before executing each operation). If the flag is set, then the local flag f_t is unset. Then the thread waits for the completion of the growing operation, or helps with migrating the table depending on the current growing strategy. After setting f_G, the growing thread t_G waits until all busy flags have been unset at least once before starting the migration. When the migration is completed, the growing flag is reset, signaling to the waiting threads that they can safely continue their table-operations. Because this protocol ensures that no thread is accessing the previous table after the beginning of the migration, it can be freed without using reference counting.

We call this method semi-synchronized, because grow and update operations are disjoint. Threads participating in one growing step still arrive asynchronously, e.g., when the parent

application called a hash table operation. Compared to the marking based protocol, we save cost during migration by avoiding CAS operations. However, this is at the expense of setting the busy flags for *every* operation. Our experiments indicates that overall this is only advantageous for updates using atomic operations like fetch-and-add that cannot coexist with the asynchronous consistency method's marking bit per element.

Overview of the Resulting Methods. In the beginning of this section, we identified two orthogonal problems that have to be solved to migrate hash tables: Which threads execute the migration? and How can we make sure that copied elements cannot be changed? For each of these problems, we formulated two strategies. The table can either be migrated by user-threads that execute operations on the table (u) or by using a pool of threads that is only responsible for the migration (p). To ensure that copied elements cannot be changed, we propose to mark elements before they are copied, thus proceeding fully asynchronously (a); and we explain a semi-synchronized protocol that ensures that all running update operations finish before the table is migrated (s).

All strategies can be combined—creating the following four growing hash table variants: *uaGrow* uses enslavement of user threads and asynchronous marking for consistency; *usGrow* also uses user threads for the migration, but ensures consistency by synchronizing updates and growing routines; *paGrow* uses a pool of dedicated migration threads for the migration and asynchronous marking of migrated entries for consistency; and *psGrow* combines the use of a dedicated thread pool for migration with the semi-synchronized exclusion mechanism.

5.4 Deletions

In Section 4, we describe why the normal folkloreHT table cannot support true deletions. Instead, we use *tombstones* to mark deleted elements, without reclaiming the cell for future insertions. This method can be improved by using our migration technique to clean the table once it is filled with too many *tombstones*.

As described in Section 4 there are two problems with this kind of deletion. The initial size of the table has to be set according to the number of overall insertions, and the amount of generated *tombstones* clutters the table slowing down future operations. Both of these problems can be solved by migrating all non-tombstone elements into a new table. The decision when to migrate the table should be made solely based on the number of insertions I since the last table migration (= *number of nonempty cells*). The count of all non-deleted elements $I - D$ is then used to decide whether the table should grow, keep the same size (notice $\gamma = 1$ is a special case for our optimized migration), or shrink. Either way, all tombstones can be removed in the course of the element migration. We implemented both a growing migration, and a migration to a table with the same size. We did not implement the shrinking mechanism.

5.5 Bulk Operations

Contention can be a huge problem for concurrent hash tables, especially in update heavy work loads. Bulk operations can also improve the construction of a hash tables from a set of elements.

Building a hash table for n elements passed to the constructor can be parallelized using integer sorting by the hash function value. This works in time $O(n/p)$ regardless how many times an element is inserted, i.e., sorting circumvents contention. See the work of Müller et al. [26] for a discussion of this phenomenon in the context of aggregation.

Processing batches of size $m = \Omega(n)$ in a globally synchronized way can use the following strategy. We outline it for the case of bulk insertions. Generalization to deletions, updates, or mixed batches is straightforward: Integer sort the elements to be inserted by their hash key in expected time $O(m/p)$. Among elements with the same hash value, remove all but the last. Then "merge"

the batch into the hash table (the hash table may have to be migrated beforehand to provide space for new elements). We can adapt ideas from parallel merging [10]. We co-partition the sorted insertion array and the hash table into corresponding pieces of size $O(m/p)$. Most of the work can now be done on these pieces in an embarrassingly parallel way—each piece of the insertion array is scanned sequentially by one thread. Consider an element $\langle x, a\rangle$ and previous insertion position i in the table. Then, we start looking for a free cell at position $\max(h(x), i)$. Atomics only have to be used for the first cluster of one partition, and once insertions enter the partition of another thread. This only happens for the last cluster of one threads partition (constant expected cluster length).

5.6 Restoring the Full Key Space

Our table uses two special keys, the empty key (`empty_key`) and the deleted key (`del_key`). Elements that actually have these keys cannot be stored in the hash table. To fix this, one could use two additional cells (c_e, and c_d) in the global hash table data structure. If the element with the empty key is inserted, then it is stored in c_e (deleted key is stored in c_d). These two cells are not accessible through linear probing, therefore, they cannot be filled with any other elements. The case distinction needed to access these cells when an insert or lookup uses one of these special keys should have rather low impact on the overall performance.

One of our growing variants (asynchronous) uses a marker bit in its key field. This halves the possible key space from 2^{64} to 2^{63}. To regain the lost key space, we can store the lost bit implicitly. Instead of using one hash table that holds all elements, we use the two subtables t_0 and t_1. The subtable t_0 holds all elements whose keys do not have their topmost bit set. While t_1 stores all elements whose keys do have the topmost bit set. Instead of storing the full keys, t_1 only stores the lower 63 bits of the keys. The topmost bit is removed (stored implicitly).

Each element can still be found in constant time, because when looking for a certain key, it is immediately obvious in which table the corresponding element will be stored. After choosing the right table, comparing the 63 explicitly stored bits can uniquely identify the correct element. Notice that both empty keys have to be stored distinctly (as described above). The size of both subtables can be chosen independently. In a balanced scenario, growing both tables at the same time could lead to improved performance.

5.7 Complex Key and Value Types

Using CAS instructions to change the content of hash table cells makes our data structure fast but limits its use to cases where keys and values fit into memory words. Lifting this restriction is bound to have some impact on performance, but we want to outline ways to keep this penalty small. The general idea is to replace the keys and or values by references to the actual data. This is a very common technique, but we feel that there are some aspects to our overall design that may have some positive impact on these techniques.

Complex Keys. There are three problems that pointers introduce to the data-structure: First, each key comparison generally costs one cache miss; second, deallocation in concurrent scenarios causes problems; and third, memory allocation costs performance.

To make things more concrete, we outline a way where the keys are strings and the hash table data structure itself manages space for the keys. When an element $\langle s, a\rangle$ is inserted, space for string s is allocated. The hash table stores $\langle r, a\rangle$ where r is a pointer to s. Unfortunately, we get a considerable performance penalty during table operations, because looking for an element with a given key now has to follow this indirection for every key comparison—effectively destroying the advantage of linear probing over other hashing schemes with respect to cache efficiency. This overhead can be reduced by two measures: First, we can make the table bigger thus reducing the

necessary search distance—considering that the keys are large anyway, this has a relatively small impact on overall storage consumption. A more sophisticated idea is to store a *signature* of the key in some unused bits of the reference to the key (on modern operating systems pointers actually only use 48 bits). This signature can be obtained from the master hash function h extracting bits that were *not* used for finding the position in the table (i.e., the least significant digits). While searching for a key y, one can then first compare the signatures before actually making a full key comparison that involves a costly pointer dereference.

Deletions do *not* immediately deallocate the space for the key, because concurrent operations might still be scanning through them. The space for deleted keys can be reclaimed when the array grows or shrinks. At that time, our migration protocols make sure that no concurrent table operations are going on.

The memory management is challenging, since we need high throughput allocation for very fine grained variable sized objects and a kind of garbage collection. On the positive side, we can find all the pointers to the strings using the hash function. All in all, these properties might be sufficiently unique that a carefully designed special purpose implementation is faster than currently available general purpose allocators. We outline one such approach: New strings are allocated into memory pages of size $\Omega(p)$. Each thread has one current page that is only used locally for allocating short strings (this page is stored using the handle described in Section 5.1). Long strings are allocated using a general purpose allocator. When the local page of a thread is full, the thread allocates a fresh page and remembers the old one on a stack. This already solves the problem in the absence of deletions. If there are deletions, then deleted strings can be removed during table migrations and used pages can be defragmented (garbage collection). This can be parallelized on a page by page basis. Each thread uses two of its partially filled pages A and B at a time. B is scanned and the strings stored there are moved to A (updating their pointer in the hash table). When A runs full, B replaces A, and a new partially filled page replaces B. When B runs empty, it is freed and the next partially filled page is chosen to replace B.

Complex Values. We can take a similar approach as for complex keys—the hash table data structure itself allocates space for complex values. This space is only deallocated during migration/cleanup phases that make sure that no concurrent table operations are affected. The find-operation only hands out *copies* of the values so that there is no danger of stale data. There are now two types of update operations. One that modifies part of a complex value using an atomic CAS operation and one that allocates an entirely new value object and performs the update by atomically setting the value-reference to the new object. Unfortunately it is not possible to use both types concurrently.

Complex Keys and *Values.* Of course, we can combine the two approaches described above. However in that case, it will be more efficient to store a single reference to a combined key-value object together with a signature.

6 USING HARDWARE MEMORY TRANSACTIONS

The biggest difference between a concurrent table, and a sequential hash table is the use of atomic processor instructions. We use them for accessing and modifying data that is shared between threads. An additional way to achieve atomicity is the use of hardware transactional memory synchronization introduced recently by Intel and IBM. The new instruction extensions can group many memory accesses into a single transaction. All changes from one transaction are committed at the same time. For other threads they appear to be atomic. General purpose memory transactions do not have progress guarantees (i.e., can always be aborted), therefore they require a fall-back path implementing atomicity (a lock or an implementation using traditional atomic instructions).

We believe that transactional memory synchronization is an important opportunity for concurrent data structures. Therefore, we analyze how to efficiently use memory transactions for our concurrent linear probing hash tables. In the following, we discuss which aspects of our hash table can be improved by using restricted transactional memory implemented in Intel Transactional Synchronization Extensions (Intel TSX).

We use Intel TSX by wrapping sequential code into a memory transaction. Since the sequential code is simpler (e.g., less branches, more freedom for compiler optimizations) it can outperform inherently more complex code based on (expensive 128-bit CAS) atomic instructions. As a transaction fall-back mechanism, we employ our atomic variants of hash table operations. Replacing the insert and update functions of our specialized growing hash table with Intel TSX variants increases the throughput of our hash table by up to 7% (see Section 8.4). Speedups like this are easy to obtain on workloads without contentious accesses (simultaneous write accesses on the same cell). Contentious write accesses lead to transaction aborts, which have a higher latency than the failure of a CAS. Our atomic fall-back minimizes the penalty for such scenarios compared to the classic lock-based fall-back that causes more overhead and serialization. In addition to simple insert or update operations, one could also use Intel TSX transactions to speed-up the cluster migration process. We did not do this here, since the migration is already optimized to use less atomics. An interesting question that would have to be solved is how many elements to migrate within one transaction.

Another aspect that can be improved through the use of memory transactions is the key and value size. On current x86 hardware, there is no atomic instruction that can change words bigger than 128 bits at once. The amount of memory that can be manipulated during one memory transaction can be far greater than 128 bits. Therefore, one could easily implement hash tables with complex keys and values using transactional memory synchronization. However, using atomic functions as fall-back will not be possible. Solutions with fine-grained locks that are only needed when the transactions actually fail, are still possible.

With general purpose memory transactions, it is even possible to atomically change multiple values that are not stored consecutively. Therefore, it is possible to implement a hash table that separates the keys from the values storing each in a separate table. In theory this could improve the cache locality of linear probing.

Overall, transactional memory synchronization can be used to improve performance and to make the data structure more flexible.

7 IMPLEMENTATION DETAILS

Bounded Hash Tables. All of our implementations are constructed around a highly optimized variant of the circular bounded *folkloreHT* that was describe in Section 4. The main performance optimizations were to restrict the table size to powers of two—replacing expensive modulo operations with fast bit operations. When initializing the capacity c, we compute the lowest power of two, that is still at least twice as large as the expected number of insertions ($2n \leq size \leq 4n$).

We also built a second non-growing hash table variant called *tsxfolkloreHT*, this variant avoids the usual CAS-operations that are used to change cells. Instead tsxfolkloreHT uses Intel TSX transactions to change elements in the table atomically. As described in Section 6, we use our usual atomic operations as fallback in case an Intel TSX transaction is aborted.

Growing Hash Tables. All of our growing hash tables use folkloreHT or tsxfolkloreHT to represent the current status of the hash table. When the table is approximately 60% filled, a migration is started. With each migration, we double the capacity. The migration works in blocks of the size 4096. Blocks are migrated with a minimum amount of atomics by using the cluster migration described in Section 5.3.1.

Table 1. Overview over Table Functionalities

Name	Plot	Std. interface	Growing	Atomic updates	Deletion	Arbitrary types
xyGrow						
uaGrow	■	using handles	✓	✓	✓	
usGrow	■	using handles	✓	✓	✓	
paGrow	◧	using handles	✓	✓	✓	
psGrow	◧	using handles	✓	✓	✓	
Junction						
linear	●	qsbr function	✓	only overwrite	✓	
grampa	◒	qsbr function	✓	only overwrite	✓	
leapfrog	◐	qsbr function	✓	only overwrite	✓	
TBB						
hash map	★	✓	✓	✓	✓	✓
unordered	★	✓	✓	✓	unsafe	✓
Cuckoo	⅄	✓	slow	✓	✓	✓
Folly	+	✓	const factor	✓		
RCU						
urcu	×	register thread	very slow	✓	✓	✓
qsbr	×	qsbr function	very slow	✓	✓	✓
FolkloreHT	◆	✓		✓		
Shunhash	◆	sync phases		partially*	✓	
Hopscotch	▲	✓		set interface	✓	
Lea Hash	▼	✓		set interface	✓	

*There are some specialized operations (that have to be chosen at the time of construction).

We use a user-space memory pool implemented in Intel Thread Building Blocks [33] to prevent a slow down due to the re-mapping of virtual to physical memory (protected by a coarse lock in the Linux kernel). This improves the performance of our growing variants, especially when using more than 24 threads. By allocating memory from this memory pool, we ensure that the virtual memory that we receive is already mapped to physical memory, bypassing the kernel lock.

All of our growing hash table variants (uaGrow, usGrow, paGrow, and psGrow) can also be instantiated using the Intel TSX based non-growing table tsxfolkloreHT as a basis.

8 EXPERIMENTAL EVALUATION

We performed a large number of experiments to investigate the performance of different concurrent hash tables in a variety of circumstances (an overview over all tested hash tables can be found in Table 1). We begin by describing the tested competitors (Section 8.1, our variants are introduced in Section 7), the test environment (Section 8.2), and the test instances (Section 8.3). Then Section 8.4 discusses the actual measurements. In Section 8.5, we conclude the section by summarizing our experiments and reflecting how different generalizations affect the performance of hash tables.

8.1 Competitors

To compare our implementation to the current state of the art, we use a broad selection of other concurrent hash tables. These tables were chosen on the basis of their popularity in applications and academic publications. We split these hash table implementations into the three categories

based on their ability to grow. Additionally, we introduce one symbol for each hash table. This symbol is used to refer to the implementation in all texts, as well as the plots.

8.1.1 Efficiently Growing Hash Tables. This group contains all hash tables, that are able to grow efficiently from a very small initial size. They are used in our growing benchmarks, where we initialize tables with an initial size of 50,000 thus making growing necessary.

Junction Linear ●*, Junction Grampa* ◒*, and Junction Leapfrog* ◐. The junction library consists of three different variants of a dynamic concurrent hash table. It was published by Preshing on github [34] after our first publication on the subject [19]. There are no scientific publications, but in his blog [35] Preshing writes some insightful posts on his implementation. In theory, junction's hash tables use an approach to growing that is similar to ours. A filled bounded hash table is migrated into a newly allocated bigger table. Although they are constructed from a similar idea, the execution seems to differ quite significantly. The junction hash tables use a *quiescent-state based reclamation* (QSBR) protocol, for memory reclamation. Using this protocol, to reclaim freed hash table memory, the user has to regularly call a designated function.

Contrary to other hash tables, we used the provided standard hash function (avalanche), because junction assumes its hash function to be invertible. Therefore, the hash function that is used for all other tables (see Section 8.3) is not usable.

The different hash tables within junction all perform different variants of open addressing. These variants are described in more detail, in one of Preshing's blogposts (see Reference [35]).

tbbHM ★ *and tbbUM* ★. (correspond to the TBB hash tables `tbb::concurrent_hash_map` and `tbb::concurrent_unordered_map`, respectively) The Threading Building Blocks [33] (TBB) library (Version 4.3 Update 6) developed by Intel is one of the most widely used libraries for shared memory concurrent programming. The two different concurrent hash tables it contains behave relatively similar in our tests. Therefore, we sometimes only plot the results of tbbHM ★. But they have some differences concerning the locking of accessed elements. Therefore, they behave very differently under contention.

cuckoo ⅄. (`cuckoohash_map`) This hash table using (bucket) cuckoo hashing as its collision resolution method, is part of the small libcuckoo library (Version 1.0). It uses a fine grained locking approach presented by Li et al. [16] to ensure consistency. Cuckoo is mentionable for their interesting interface, which combines easy container style access with an update routine similar to our update interface.

8.1.2 Hash Tables with Limited Growing Capabilities. This group contains all hash tables that can only grow by a limited amount (constant factor of the initial size) or become very slow when growing is required. When testing their growing capabilities, we usually initialize these tables with half their target size. This is comparable to a workload where the approximate number of elements is known but cannot be bound strictly.

folly +. (`folly::AtomicHashMap`) This hash table was developed at facebook as a part of their open source library folly [28, 29] (Version 57:0). It uses restrictions on key and data types similar to folkloreHT. In contrast to our growing procedure, the folly table grows by allocating additional hash tables. This increases the cost of future queries. The implementation has a built in maximal growing factor of ≈18 ($\times$*initialsize*).

RCU ×*/RCU QSBR* ×. This hash table is part of the Userspace RCU library (Version 0.8.7) [32], that brings the read copy update principle to userspace applications. Read copy update is a set of protocols for concurrent programming, that are popular in the Linux kernel community [23]. The

hash table uses split-ordered lists to grow in a lock-free manner [36]. RCU uses the recommended read-copy-update variant (`urcu`). RCU QSBR uses a QSBR based protocol that is comparable to the one used by junction hash tables. It forces the user to repeatedly call a function with each participating thread. We tested both variants, but in many plots we show only RCU ×, because both variants behaved very similarly in our tests.

Both tables become very slow when growing becomes necessary. To lower the time needed for our tests, we put the tables into the semi growing category (where tables never grow by more than a factor of two).

8.1.3 Non-Growing Hash Tables. One of the most important subjects of this publication is offering a scalable asynchronous migration for the simple folkloreHT. While this makes it usable in circumstances where bounded tables cannot be used, we want to show that even when no growing is necessary, we can compete against bounded hash tables. Therefore, it is reasonable to use our growing hash table even in applications where the number of elements can be bounded in a reasonable manner, offering a graceful degradation in edge cases and allowing improved memory usage if the bound is not reached.

FolkloreHT ◆. Our implementation of folkloreHT described in Section 4. Notice that this hash table is the core of our growing variants. Therefore, we can immediately determine the overhead that the ability for growing places on this implementation (overhead for approximate counting and shared pointers).

Shunhash ⧫. This hash table implementation proposed by Shun and Blelloch [37] is designed to be deterministic when used in a phase concurrent manner, i.e., no reads can occur concurrently with writes. There is also a nondeterministic variant that we used for our tests.

Hopscotch Hash ▲. Hopscotch hashing (version 2.0) is one of the more popular variants of open addressing. The version we tested, was published by Herlihy et al. [12] connected to their original publication proposing the technique. Interestingly, the provided implementation only implements the functionality of a hash set (unable to retrieve/update stored data). Therefore, we had to adapt some tests to account for that (`insert`≅`put` and `find`≅`contains`).

LeaHash ▼. This hash table is designed by Lea [15] as part of Java's Concurrency Package. We have obtained a C++ implementation, which was published together with the hopscotch table. It was previously used during for experiments by Herlihy et al. [12] and Shun and Blelloch [37]. LeaHash uses hashing with chaining, and the implementation that we use has the same hash set interface as hopscotch.

As previously described, we used hash set implementations for Hopscotch hashing, as well as LeaHash (they were published like this). They should easily be convertible into common hash map implementations, without losing too much performance, but probably using quite a bit more memory.

8.1.4 Sequential Variants. To report absolute speedup numbers, we implemented sequential variants of growing and fixed size tables. They do not use any atomic instructions or similar slowdowns. They outperform popular choices like Google's dense hash map significantly (80% increased insert throughput), making them a reasonable approximation for the optimal sequential performance.

8.1.5 Color/Marker Choice. For practicality reasons, we chose not to print a legend with all of our figures. Instead, we use this section to explain the color and marker choices for our plots (see Section 8.1 and Table 1), hopefully making them more readable.

Some of the tested hash tables are part of the same library. In these cases, we use the same marker, for all hash tables within that library. The different variants of the hash table are then differentiated using the line color (and filling of the marker).

For our own tables, we mostly use ■ and ■ for uaGrow and usGrow, respectively.

8.2 Hardware Overview

Most of our experiments were run on a two socket machine, with Intel Xeon E5-2683 v4 processors (previously codenamed Broadwell). Each processor has 16 cores running at 2.1Ghz base frequency. The two sockets are connected by two Intel QPI-links. Distributed to the two sockets there are 512GB of main memory (256GB each). The processors support Intel Hyper-Threading, AVX2, and Intel TSX technologies.

This system runs a Ubuntu distribution with the kernel number 4.4.0-109-generic. We compiled all our tests with gcc 7.3.0—using optimization level `-O3` and the necessary compiler flags (e.g., `-mcx16`, `-msse4.2`, among others).

Additionally, we executed some experiments on a 32-core 4-socket Intel Xeon E5-4640 (SandyBridge-EP) machine, with 512GB main memory, to verify our findings, and show improved scalability even on four-socket machines.

8.3 Test Methodology

Each test measures the time it takes, to execute 10^8 hash table operations (*strong scaling*). Each data point was computed by taking the average of five separate execution times. Different tests use different hash table operations and key distributions. The used keys are pre-computed before the benchmark is started. Each speedup given in this section is computed as the *absolute speedup* over our hand-optimized sequential hash table.

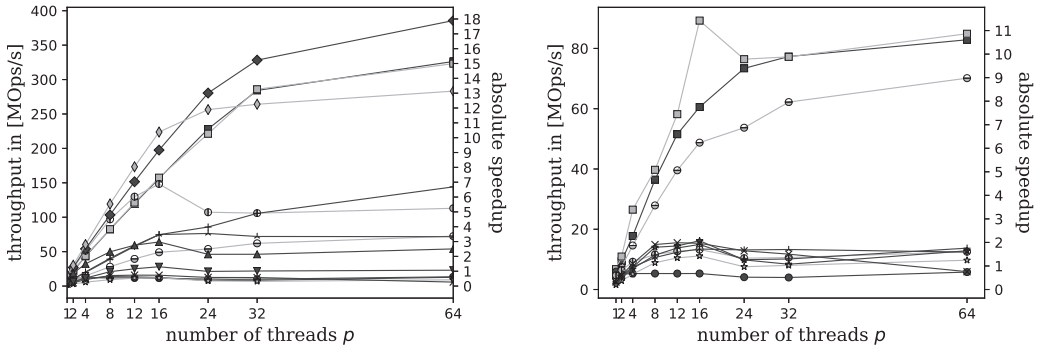
The work is distributed among threads dynamically. While there is work to do, threads reserve blocks of 4,096 operations to execute (using an atomic counter). This ensures a minimal amount of work imbalance, making the measurements less prone to variance.

All random inputs are precomputed before the execution, they are scattered between all participating NUMA nodes (they are generated by the threads that perform the test). Two executions of the same test will always use the same input keys. Most experiments are performed with uniformly random generated keys (using the Mersenne twister random number generator [21]). Since real-world inputs may have recurring elements, there can be contention, which can potentially lead to performance issues. To test hash table performance under contention, we use Zipf's distribution to create skewed key sequences. Using the Zipf distribution, the probability for any given key k is $P(k) = 1/(k^s \cdot H_{N,s})$, where $H_{N,s}$ is the Nth generalized harmonic number $\sum_{k=1}^{N} \frac{1}{k^s}$ (normalization factor) and N is the universe size ($N = 10^8$). The exponent s can be altered to regulate the contention. We use the Zipf distribution, because it closely models some real world inputs like natural language, natural size distributions (e.g., of firms or internet pages), and even user behavior [1, 2, 4]. Notice that key generation is done prior to the benchmark execution as to not influence the measurements unnecessarily (this is especially necessary, for skewed inputs).

As a hash function, we use two CRC32C x86 instructions with different seeds, to generate the upper and lower 32 bits of each hash value. Their hardware implementation minimizes the computational overhead.

8.4 Experiments

The most basic functionality of each hash table is inserting and finding elements. The performance of many parallel algorithms depends on the scalability of parallel insertions and finds. Therefore,



(a) Insert into pre-initialized table (b) Insert into growing table

Fig. 2. Throughput while inserting 10^8 elements into a previously empty table (see Table 1 for the legend).

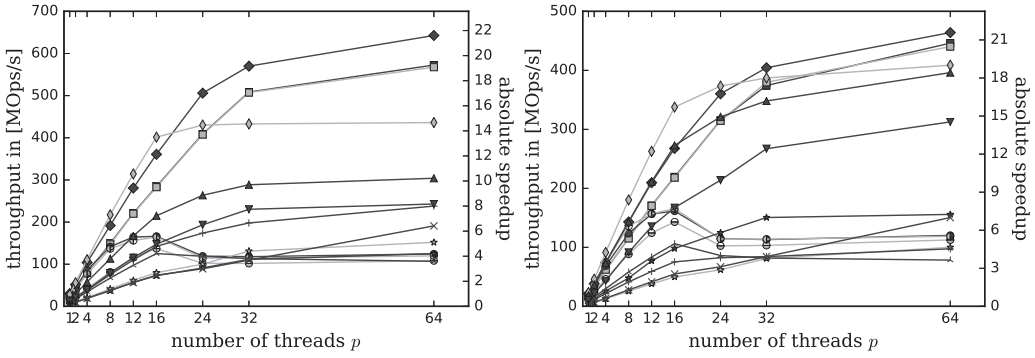
we begin our experiments with a thorough investigation into the scalability of these basic hash table operations.

Insert Performance. We begin with the very basic test of inserting 10^8 different uniformly random keys, into a previously empty hash table. For this first test, all hash tables have been initialized to the final size making growing unnecessary. The results presented in Figure 2(a) show that for large thread counts folkloreHT ◆ is optimal in this case. Since there is no migration necessary, and the table can be initialized large enough, such that long search distances become very improbable. The large discrepancy between folkloreHT ◆, and all previous growable hash tables is what motivated us, to work with growable hash tables in the first place. As shown in the plot, our growing hash tables uaGrow ■ and usGrow ■ lose about 16% of performance over folkloreHT ◆ (18× Speedup vs. 15×). This performance loss can be explained with some overheads that are necessary for eventually growing the table (e.g., estimating the number of elements). All hash tables that have a reasonable performance (>30% of folkloreHT ◆ performance), are variants of open addressing, first, is shunhash ⧫ 13 this is not surprising, since shunhash is very similar in functionality to folkloreHT ◆. Afterwards, folly +6.6 and junction leapfrog ◐ 5.5 at $p = 16$, which also have similar restrictions on key and value types. All hash tables that can natively handle generic data types (without using pointers) are severely outclassed (★, ★, ⅄, ×, and ×).

After this introductory experiment, we take a look at the growing capability of each table. We again insert 10^8 elements into a previously empty table. This time, the table has only been initialized, to hold 50,000 elements ($5 \cdot 10^7$ for all tables we identified as *semi growing*). We can clearly see from the plots in Figure 2(b), that our hash table variants are significantly faster than any comparable tables. The difference becomes especially obvious once two sockets are used (>16 cores). With more than one socket, none of our competitors could achieve any significant speedups. On the contrary, many tables become slower when executed on more cores. This effect does not happen for our table.

Junction grampa ◒ is the only other growing hash table that achieves absolute speedups higher than 2. All other growing hash tables are severely outperformed by our hash table usGrow ■. Compared to all other tables, we achieve at least seven times the performance (descending order; using 64 threads) of folly + (6.2×), tbb hm ★ (6.6×), junction leapfrog ◐ (6.8×), cuckoo ⅄ (6.8×), tbb um ★ (8.7×), rcu qsbr × (13×), rcu × (14×), and junction linear ● (15×).

The absolute speedups we reach in this benchmark are ≈11× compared to the sequential version (also with growing). This is worse then the absolute speedup in the non-growing test (≈15), suggesting that our migration is nearly as scalable as hash table accesses. Overall our growing



(a) Successful finds (b) Unsuccessful finds

Fig. 3. Performance and scalability calling 10^8 unique find operations, on a table, containing 10^8 unique keys (see Table 1 for the legend).

hash table performs nearly as good as folkloreHT ◆ in the non-growing case, while performing similarly well in tests where growing is necessary.

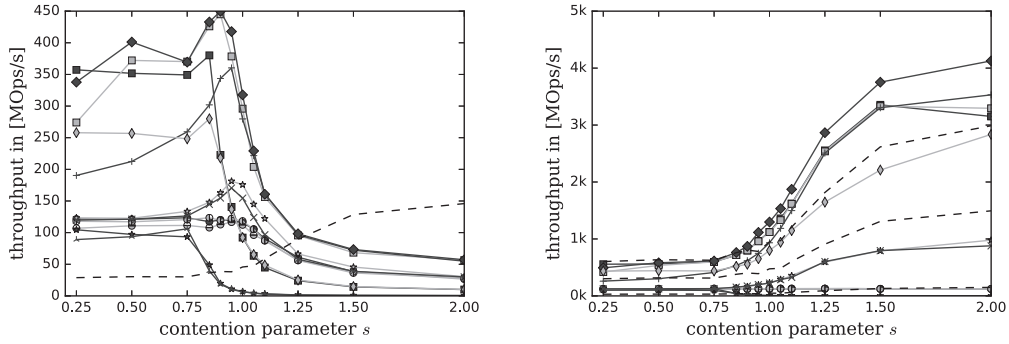
Find Performance. When looking for a key in a hash table there are two possible outcomes, either it is in the table or it is not. For most hash tables not finding an element takes longer than finding said element. Therefore, we present two distinct measurements for both cases Figure 3(a) and Figure 3(b). We measure the performance of successful finds by looking for all 10^8 elements, that were previously inserted into the hash table. For the unsuccessful measurement, 10^8 uniformly random keys are searched in the same hash table.

All the measurements made for these plots were done on a preinitialized table (preinitialized before insertion). This actually makes a difference for many hash tables later in the article (Memory Test), we will see that our find performance is actually better on a grown table. For some hash tables, the opposite is the case. Hash tables that grow by allocating additional tables have worse find performance on a grown table, as they can have multiple active tables at the same time (all of them have to be checked).

Obviously, find workloads achieve bigger throughputs than insert heavy workloads – no memory is changed and no coordination is necessary between processors (i.e., atomic operations). According to these tests, find operations seem to scale better with multiple processors than insert operations. Here, our growable implementations achieve speedups of 20 (unsuccessful)/ 19 (successful) compared to 15 in the insertion case.

When comparing the find performance between different tables, we can see that the performance advantage of our implementations (and shunhash ⧫) over other implementations with open addressing is not as wide as in the insert test. Hopscotch hashing ▲ performs especially well in the unsuccessful case, where it reaches the performance of folkloreHT for lower thread counts. However, this has to be taken with a grain of salt, because the tested implementation only offers the functionality of a hash set (contains instead of find). Therefore, less memory is needed per element and more elements can be hashed into one cache line, making lookups significantly more cache efficient.

For our hash tables, the performance reduction between successful and unsuccessful finds is around 22 to 26%. The difference of absolute speedups between both cases is relatively small—suggesting that sequential hash tables suffer from the same performance penalties. The biggest difference has been measured for folly + (42 to 59% reduced performance for unsuccessful finds).



(a) Update – Overwrite (the dashed line shows the unscaled sequential performance)

(b) Successful Finds (the dashed lines show sequential performance multiplied by 1×, 10× and 20× – for scaling)

Fig. 4. Throughput executing 10^8 operations using a varying amount of skew in the key sequence (all keys were previously inserted; using 64 threads; see Table 1 for the legend).

Later, we see that the reason for this is likely that folly + is configured to use only relatively little memory (see Figure 10). When initialized with more memory, its performance gets closer to the performance of other hash tables using open addressing.

Performance under Contention. Up to this point, all data sets we looked at contained uniformly random keys sampled from the whole key space. This is not necessarily the case in real world data sets, some keys might appear many times. One key might even dominate the input. Access to this key's element can slow down the global progress significantly, especially if hash table operations use (fine grained) locking, to protect hash table accesses.

To benchmark the robustness of the compared hash tables on these degenerate inputs, we construct the following test setup. Before the execution, we compute a sequence of skewed keys using the Zipf distribution described in Section 8.3 to 10^8 keys from the range $1..10^8$ with a varying exponent s (amount of skew). Then the table is filled with all keys from the same range $1..10^8$.

For the first benchmark, we execute an update operation for each key of the skewed key sequence, overwriting its previously stored element (Figure 4(a)). These update operations will create contentious write accesses to the hash table. Note that updates perform simple overwrites, i.e., the resulting value of the element is not dependent on the previous value. The hash table will remain at a constant size for the whole execution, making it easy to compare different implementations independent of effects introduced through growing. In the second benchmark, we execute find operations instead of updates, thus creating contending read accesses.

For sequential hash tables, contention on some elements can have very positive effects. When one cell is visited repeatedly, its contents will be cached and future accesses will be faster. The sequential performance is shown in our figures using a dashed black line. For concurrent hash tables, contention has very different effects.

Unsurprisingly, the effects experienced from contention are different between writing and reading operations. The reason is that multiple threads can read the same value simultaneously, but only one thread at a time can change a value (on current CPU architecture). Therefore, read accesses can profit from cache effects—much like a sequential hash table, while write accesses are hindered by the contention. This goes so far, that for workloads with high contention, no concurrent hash table can achieve the performance of a sequential table.

Apart from slowdown because of exclusive write accesses, there is also the additional problem of cache invalidation. When a value is repeatedly changed by different cores of a multi-socket architecture, cached copies have to be invalidated whenever this value is changed. This leads to bad cache efficiency and also to high traffic on QPI Links (connections between sockets).

From the update measurement shown in Figure 4(a), it is clearly visible that the serious impact through contention begins between $s = 0.85$ and 0.95. Up until that point, contention has a positive effect even on update operations. For a skew between $s = 0.85$ and 0.95, about 1% to 3% of all accesses go to the most common element (key k_1). This is exactly the point where $1/p \approx P(k_1)$, therefore, on average there will be one thread changing the value of k_1.

On these contentious update operations the difference between usGrow ■ and uaGrow ■ becomes visible for the first time. The reason for this is that uaGrow ■ uses 128-bit CAS operations to update elements while simultaneously making sure, that the marked bit of the element has not been set before the change. This can be avoided using the usGrow ■ variant by specializing the update method to use atomic operations on the data part of the element. This is possible, because updates and grow routines cannot overlap in this variant.

The plot in Figure 4(b) shows that concurrent hash tables achieve performance improvements similar to sequential ones when repeatedly accessing the same elements. Our hash table can even increase its speedups over uniform access patterns, the highest speedup of folkloreHT ◆ is 32 at $s = 1.25$. Since the speedup is this high, we also included scaled plots showing 10× and 20×. Our growing variants improve comparably well with a speedup of 28 both for uaGrow ■ and usGrow ■ (at $s = 1.25$).

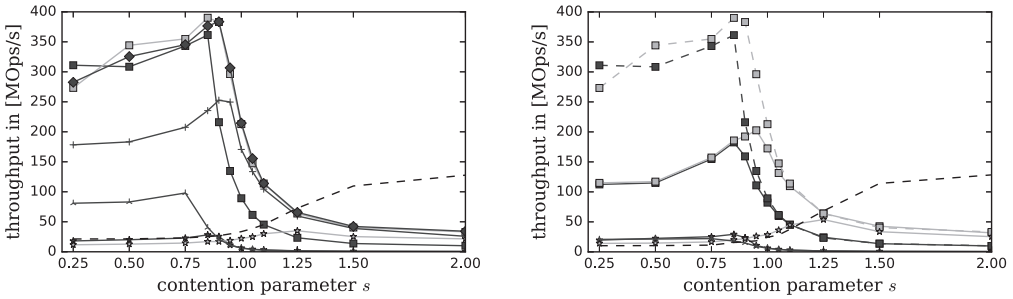
Overall, we see that our folkloreHT ◆ implementation outperforms all other competitors. Our growable variant usGrow ■ is consistently close to folkloreHT's performance—outperforming all hash tables that have the ability to grow.

Aggregation—A Common Use Case. Hash tables are often used for key aggregation. The idea is that all data elements connected to the same key are aggregated using a commutative and associative function. For our test, we implemented a simple key count program. To implement the key count routine with a concurrent hash table, an insert-or-increment function is necessary. For some tables, we were not able to implement an update function, where the resulting value depends on the previous value, within the given interface (junction tables, rcu tables, shunhash, hopscotch, and leahash). This was mainly a problem of the used interfaces, therefore, it could probably be solved by reimplementing a more functional interface. For our table this can easily be achieved with the `insertOrUpdate` interface using an increment as update function (see Section 4).

The aggregation benchmark uses the same Zipf key distribution as the other contention tests. For 10^8 skewed keys, the insert-or-increment function is called. Contrary to the previous contention test, there is no pre-initialization. Therefore, the number of distinct elements in the hash table is dependent on the contention of the key sequence (given by s). This makes growable hash tables even more desirable, because the final size can only be guessed before the execution.

Like in previous tests, we make two distinct measurements. One with growing (Figure 5(a)) and one without (Figure 5(b)). In the test without growing, we initialize the table with a size of 10^8 to ensure that there is enough room for all keys, even if they are distinct. We excluded the semi-growing tables from Figure 5(b) as approximating the number of unique keys can be difficult. To set the growing performance into relation, we show some non-growing tests. Growing actually costs less in the presence of contentious updates, because the resulting table will be smaller than without contention, therefore, fewer growing steps can be amortized over the same number of operations.

The result of this measurement is clearly related to the result of the contentious overwrite test shown in Figure 4(a). However, changing a value by increment has some slight differences to



(a) Aggregation using a pre-initialized size of 10^8 ($\Rightarrow$ *size* = |*operations*|).

(b) Aggregation with growing. Dashed plots (■ and ■) indicate non-growing performance.

Fig. 5. Throughput of an aggregation executing 10^8 insert-or-increment operations using skewed key distributions (using 64 threads; see Table 1 for the legend). The dashed black line indicates sequential performance. Some tables are not shown, because their interface does not support insert-or-increment in a convenient way.

overwriting it, since the updated value of an insert-or-increment is dependent on its previous value. In the best case, this increment can be implemented using an atomic fetch-and-add operation (i.e., usGrow ■, folkloreHT ◆, and folly +). However this is not possible for all hash tables, sometimes dependent updates are implemented using a read-modify-CAS cycle (i.e., uaGrow ■) or fine grained locking (i.e., tbb hash map ★ or cuckoo ⅄).

Until $s = 0.85$, uaGrow ■ seems to be the more efficient option, since it has an increased writing performance and the update cycle will be successful most of the time. From that point on, usGrow ■ is clearly more efficient, because fetch-and-add behaves better under contention. For highly skewed workloads, it comes really close to the performance of folkloreHT ◆, which again performs the best out of all implementations.

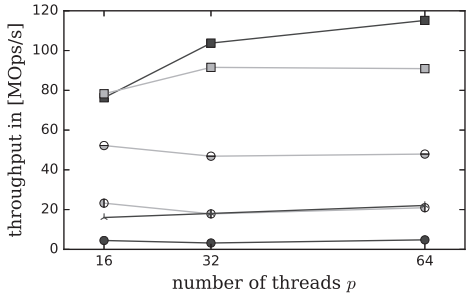
Deletion Tests. As described in Section 5.4, we use migration, not only to implement an efficiently growing hash table but also to clean up the table after deletions. This way all tombstones are removed, and thus freed cells are reclaimed. But how does this fare against different ways of removing elements? This is what we investigate with the following benchmark.

The test starts on a prefilled table (10^6 elements) and consists of 10^8 insertions—each immediately followed by a deletion. Therefore, the table remains at approximately the same size throughout the test ($\pm p$ elements). All keys used in the test are generated before the benchmark execution (uniform distribution). As described in Section 8.3, all keys are stored in one array. Each insert uses an entry from this array distributed in blocks of 4,096 from the beginning. The corresponding deletion uses the key that is 10^6 elements prior to the corresponding insert. Thus, the keys stored within the hash table are contained in a sliding window of the key array.

We constructed the test to keep a constant table size, because this allows us to test non-growing tables without significantly overestimating the necessary capacity. All hash tables are initialized with 1.5×10^7 capacity; therefore, it is necessary to reclaim deleted cells to successfully execute the benchmark.

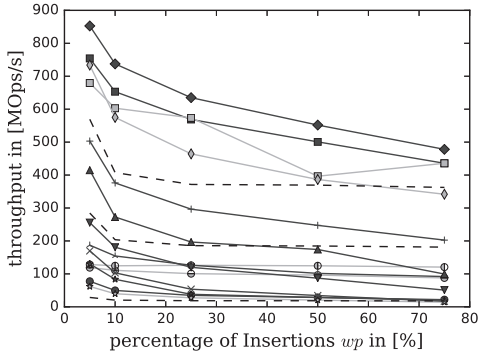
The result of this experiment clearly shows, that our implementation performs better than other available implementations. It also shows that the performance of operations does not get impacted by tombstones, and that the memory reclamation works as expected. Otherwise, we would not reach values of $100 MOps/s$ and more (note that $1 Ops$ is one insert + one remove).

Mixed Insertions and Finds. It can be argued that some of our tests are just micro-benchmarks, which are not representative of real-world workloads that often mix insertions with lookups. To

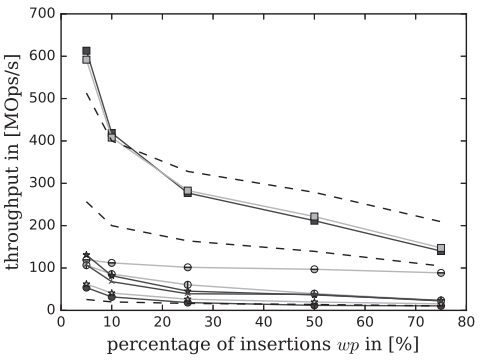


(a) By alternating between insertions and deletions, we keep the number of elements in the table at approximately 10^7 elements. For the purpose of computing the throughput, 10^8 such alternations are executed, each counting as one operation (1 Op = insert + delete).

Fig. 6. Throughput in a test using deletion. Some tables have been left out of this test, because they do not support deletion with memory reclamation (see Table 1 for the legend).



(a) Mixed insertions and finds on a pre-initialized table ($wp \cdot 10^8 + pre$).



(b) Mixed insertions and finds on a growing table.

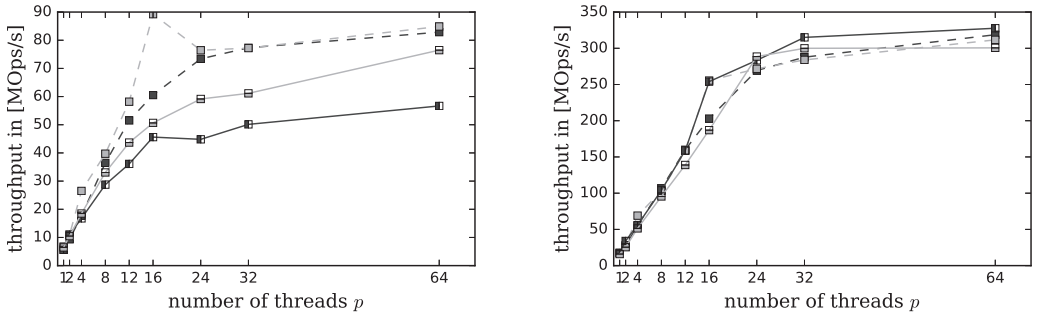
Fig. 7. Executing 10^8 operations mixed between insertions and finds (using 64 threads; see Table 1 for the legend). Dashed lines indicate sequential performance (1×, 10×, and 20×). Find keys are generated in a fair way, that ensures that most find operations are successful.

address these concerns, we want to show that mixed function workloads (i.e., combined find and insert workloads) behave similarly.

As in previous tests, we generate a key sequence for our test. Each key of this sequence is used for an insert or a find operation. Overall, we generate 10^8 keys for our benchmark. For each key, insert or find is chosen at random according to the write percentage wp. In addition to the keys used in the benchmark, we generate a small number of keys ($pre = 8192 \cdot p = 2\,\text{blocks} \cdot p$) that are inserted prior to the benchmark. This ensures that the table is not empty and there are keys that can be found with lookups.

The keys used for insertions are drawn uniformly from the key space. Looking for a random inserted element is representative of the overall distribution of probing distances in the table. It is more fair than only searching for the pre-inserted keys (very short probing distance), and searching for one of the last inserted keys (cache effects). It also means that threads will look for keys inserted by other threads. Therefore, we generate keys for find operations, by randomly selecting keys that are inserted earlier in the key sequence (before a sliding window of $8{,}192 \cdot p$ keys). Keys like that are usually already in the table when the find operation is called.

Notice that this method does not strictly ensure that all search keys are already inserted. In our practical tests, we found that the number of keys that were not found was negligible for performance purposes (usually below 10,000).



(a) Insertions into Growing Table. (b) Unsuccessful Finds after Growing.

Fig. 8. Comparison between our regular implementation and the variant using a dedicated migration thread pool (see Table 1 for the legend).

Comparable to previous tests, we test all hash tables with and without the necessity to grow the table. In the non-growing test the size of each table is pre-initialized to be $c = pre + (wp \cdot 10^8)$. In the growing tests, semi-growing hash tables are initialized with half that capacity.

Similar to previous tests, it is obvious that our non-growing linear probing hash table folkloreHT ◆ outperforms most other tables especially on find-heavy workloads. Overall, our hash tables behave similar to the sequential solution with a constant speedup around a factor of 20 to 30. Interestingly, the running time does not seem to be a linear function (over wp). Instead, the performance decreases sharply even when the ratio of insertions is small. One reason for this could be that for find-heavy workloads, the table remains relatively small throughout large parts of the execution. Therefore, cache effects and similar influences could play a role, since lookups only look for a small sample of elements that is already in the table. This seems to be similar both for sequential and the concurrent instances.

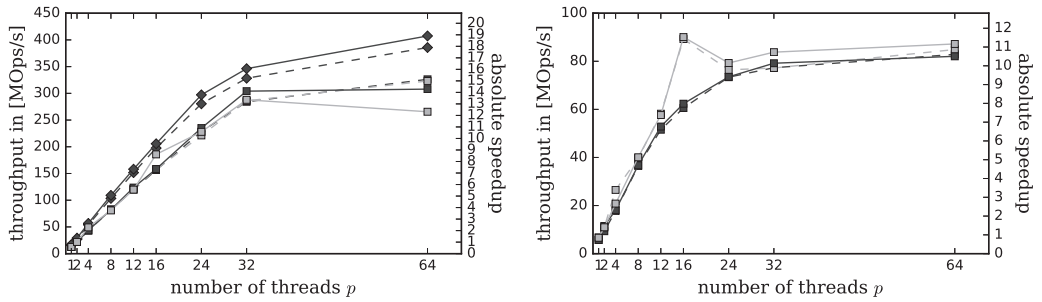
Using Dedicated Growing Threads. In Section 5.3.2 and 7, we describe the possibility of using a pool of dedicated migration threads that grow the table cooperatively. Usually the performance of this method does not differ greatly from the performance of the enslavement variant used throughout our testing. This can be seen in Figure 8. Therefore, we omitted these variants from most plots.

In Figure 8(a), we can see how much performance is lost between the the variants using a thread pool and their counterparts (uaGrow ■ ≅ paGrow ◧ and usGrow ■ ≅ psGrow ◧). The differences happen only during the migration. The reason for this slowdown is that using additional migration threads makes some communication with the operating system necessary. The growing threads have to be awoken and put to sleep for each migration cycle (scheduling and notification). A find benchmark Figure 8(b) executed after the migration does not show any significant differences between the two growing methods.

Using Intel TSX Technology. As described in Section 6, concurrent linear probing hash tables can be implemented using Intel TSX technology to reduce the number of atomic operations. Figure 9 shows some of the results using this approach.

The implementation used in these tests changes only the operations within folkloreHT to use Intel TSX-transactions. Atomic fallback implementations are used, when a transaction fails. We also instantiated our growing hash table variants, to use the Intel TSX-optimized table as underlying hash table implementation.

We tested this variant with a uniform insert workload (see “Insert Performance”), because the lookup implementation does not actually need a transaction. We also show the non-Intel TSX variant, using dashed lines, to indicate the relative performance benefits.



(a) Without growing (pre-initialized). (b) Insertions with growing.

Fig. 9. Comparison between our regular implementation and the variant using Intel TSX-transactions in place of atomics (dashed lines are regular variants without Intel TSX).

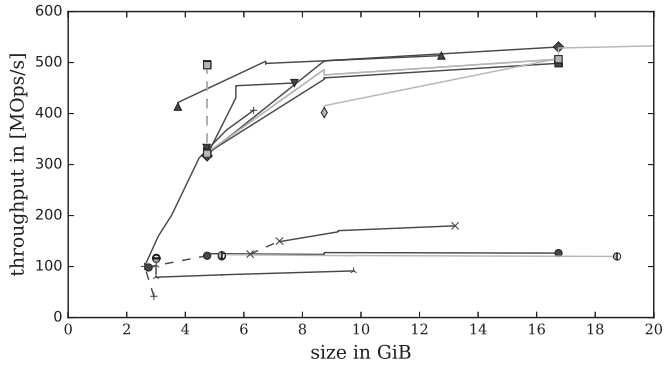
In Figure 9(a) one can see that Intel TSX-optimized hash tables offer improved performance as long, as growing is not necessary (up to 7%). Unfortunately, Figure 9(b) paints a different picture for instances where growing is necessary. While Intel TSX can be used to improve the usGrow ■ variant of our hash table especially when using hyperthreading, it offers no performance benefits in the uaGrow ■ variant. The reason for this is that the running time in these measurements is dominated by the table migration, which is not optimized for Intel TSX-transactions. We estimate that a well optimized Intel TSX-migration that actually uses transactions in the origin table can gain performance increases similar to those witnessed in the non-growing case.

Memory Consumption. One aspect of parallel hash tables that we did not talk about until now is memory consumption. Overall, a low memory consumption is preferable, but having less cells means that there will be more hash collisions. This leads to longer running times especially for non-successful find operations.

Most hash tables do not allow the user to set a specific table size directly. Instead they are initialized using the expected number of elements. We use this mechanism to create tables of different sizes. Using these different hash tables with different sizes, we find out how well any one hash table scales when it is given more memory. This is interesting for applications where the hash table speed is more important than its memory footprint (lookups to a small or medium sized hash table within an application's inner loop).

The values presented in Figure 10 are acquired by initializing the hash tables with different table capacities (50,000, 0.5×, 1.0×, 1.25×, 1.5×, 2.0×, 2.5×, 3.0×10^8 expected elements; semi- and non-growing hash tables start at 0.5× and 1×, respectively). During the test, the memory consumption is measured by logging the size of each allocation, and deallocation during the execution (done by replacing allocation methods, e.g., `malloc` and `memalign`). Measurements with growing (initial capacity $< 10^8$) are marked with dashed lines. Afterwards the table is filled with 10^8 elements. The plotted measurements show the throughput that can be achieved when doing 10^8 unsuccessful lookups on the preinitialized table. This throughput is plotted over the amount of allocated memory each hash table used.

The minimum size for any hash table should be around $1.53\,\text{GiB} \approx 10^8 \cdot (8\,\text{B} + 8\,\text{B})$ (Key and Value each have 8 B). Our hash table uses a number of cells equal to the smallest power of 2 that is at least two times as large as the expected number of elements. In this case, this means we use $2^{28} \approx 2.7 \cdot 10^8$; therefore, the table will be filled to ≈37% and use exactly 4GiB. We believe that this memory usage is reasonable, especially for heavily accessed tables where the performance is important. This is supported by our measurements as all hash tables that use less memory have bad performance.



(a) Performance of unsuccessful find operations over the size of the data structure.

Fig. 10. For these tests 10^8 keys are searched (unsuccessfully) on a hash table containing 10^8 elements. Prior to the setup of the benchmark, the tables were initialized with different sizes (there can be many points on one (x-)coordinate) (see Table 1 for the legend).

Most hash tables round the number of cells in some convenient way. Therefore, there are often multiple measurement points using the same amount of memory. As expected, using the same amount of memory will usually achieve a comparable performance. Out of the tested hash tables only the folly + hash table grows linearly with the expected final size. It is also the hash table that gains the most performance by increasing its memory. This makes a lot of sense considering that it uses linear probing and is by default configured to use more than 50% of its cells.

The plot also shows that some hash tables do not gain any performance benefits from the increased size. Most notable for this are cuckoo ⅄, all variations of junction ● ◒ ◐ and the urcu hash tables ×.

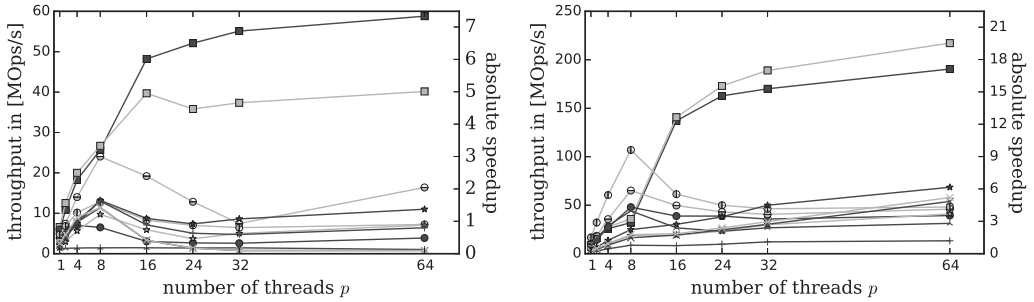
There are also some things that can be learned about growing hash tables from this plot. Our migration technique ensures that our hash table has the exact same size when growing is required as when it is preinitialized using the same number of elements. Even though the resulting size is the same, lookup operations are significantly better on tables that had at least one migration. We believe the reason for this is, that when the table is initialized, memory is moved to the NUMA node of the one node, that initializes the table. During the migration, the table is initialized by all threads cooperatively therefore scattering the memory to all NUMA nodes.

This is not true, for many of our competitors. All Junction tables and RCU produce smaller tables when growing was used, they also suffer from a minor slowdown, when using lookups on these smaller tables. The growing mechanism in folly seems to be even worse. When growing is necessary, folly builds a bigger table and still offers a worse lookup performance.

Scalability on a Four-Socket Machine. Bad performance in multi-socket scenarios is recurring theme throughout our testing. This is especially true for some of our competitors where Two-Socket running times are often worse than One-Socket running times. To further expand the understanding of this problem we ran some tests on the Four-Socket machine described in Section 8.2.

The used test instances are generated similar to the insert/find tests described in the beginning of this section (10^8 executed operations with uniformly random keys). The used random keys are distributed among all NUMA nodes that take part in the specific measurement. The results can be seen in Figure 11(a) (Insertions) and Figure 11(b) (unsuccessful finds).

Our competitor's hash tables seem to be a lot more effective when using only one of the four sockets (compared to one of two sockets on the two-socket machine). This is especially true for the Lookup workload where the junction hash tables ● start out more efficient than our implementation. However, this effect seems to invert once multiple sockets are used.



(a) Insertions into a growing hash table. (b) Unsuccessful Lookups into the filled table.

Fig. 11. Basic tests made on our Four-Socket machine, consisting of four eight core E5-4640 processors with 2.4GHz each (codenamed Sandybridge) and 512GB main memory (see Table 1 for the legend).

In the test using lookups, there seems to be a performance problem using our hash table. It seems to scale sub-optimally on one socket. On two sockets, however, the hash table seems to scale significantly better.

Overall the four-socket machine reconfirms our observations. None of our competitors scale well when a growing hash table is used over multiple sockets. On the contrary, using multiple sockets will generally reduce the throughput. This is not the case for our hash table. The efficiency is reduced when using more then two sockets but the absolute throughput at least remains stable.

8.5 The Price of Generality

Having looked at many detailed measurements, let us now try to get a bigger picture by asking which hash table performs well for specific requirements and how much performance has to be sacrificed for additional flexibility. This will give us an intuition, where performance is sacrificed on our way to a fully general hash table. Seeing that all tested hash tables fail to scale linearly on multi-socket machines, we try to answer the question if concurrent hash tables are worth their overhead at all.

At the most restricted level—no growing/deletions and word sized key and value types—we have shown that common linear probing hash tables offer the best performance (over a number of operations). Our implementation of this "folklore" solution (folkloreHT) outperforms different approaches consistently, and performs at least as good as other similar implementations (i.e., shunhash). We also showed, that this performance can be improved by using Intel TSX technology. Furthermore, we have shown that our approach to growing hash tables does not affect the performance on known input sizes significantly (preinitialized table to the correct size).

Sticking to fixed data types but allowing dynamic growing, the best data structures are our growing variants ({ua, us, pa, ps}Grow). The difference in our measurements between pool growing (paGrow and psGrow) and the corresponding variants with enslavement (uaGrow and paGrow) are not very big. Growing with marking performs better than the semi-synchronized growing except for update heavy workloads. The price of growing compared to a fixed size is less than a factor of two for insertions and updates (aggregation) and negligible for find-operations. Moreover, this slowdown is comparable to the slowdown experienced in sequential hash tables when growing is necessary. None of the other data structures that support growing come even close to the performance of our data structures. For insertions and updates we are an order of magnitude faster then many of our competitors. Furthermore, only one competitor achieves speedups above one when inserting into a growing table (junction grampa).

Among the tested hash tables, only TBB, Cuckoo, and RCU have the ability to natively store arbitrary key-/value-type combinations. All of them are at least one order of magnitude slower than our solutions ({ua, us, pa, ps}Grow). In our opinion, this restricts the use of these data structures to situations where hash table accesses are not a computational bottleneck. For more demanding applications the only way to go is to get rid of the general data types or the need for concurrent hash tables altogether. We believe that the generalizations we have outlined in Section 5.7 will be able to close this gap. Actual implementations and experiments are therefore interesting future work.

Finally, let us consider the situation where we need general data types but no growing. Again, all the competitors are an order of magnitude slower for insertion than our bounded hash tables. The single exception is cuckoo, which is only five times slower for insertion and six times slower for successful reads. However, it severely suffers from contention being an almost record breaking factor of 6 200 slower under find-operations with contention. Again, it seems that better data structures should be possible.

9 CONCLUSION

We demonstrate that a bounded linear probing hash table specialized to pairs of machine words has much higher performance than currently available general purpose hash tables like Intel TBB-, Leahash-, or RCU-based implementations. This is not surprising from a qualitative point of view given previous publications [13, 37, 39]. However, we found it surprising how big the differences can be in particular in the presence of contention. For example, the simple decision to require a lock for reading can decrease performance by almost four orders of magnitude.

Perhaps our main contribution is to show that integrating an adaptive growing mechanism into that data structure has only a moderate performance penalty. Furthermore, the used migration algorithm can also be used to implement deletions in a way that reclaims freed memory. We also explain how to further generalize the data structure to allowing more general data types.

The next logical steps are to implement these further generalizations efficiently and to integrate them into an easy to use library that hides most of the variants from the user, e.g., using programming techniques like partial template specialization. It would also be possible to eliminate the small amount of locking introduced by our migration techniques. Currently, the hash table might stall, if the operating system swaps a thread that is currently helping with the migration. This could be avoided using hazard pointers and a different block migration.

Further directions of research could be to look into translating the same low-communication migration mechanism to different hashing schemes (i.e., cuckoo hash tables similar to Reference [18] or hopscotch hash tables). It could even be applied to other types of data structures, such as concurrent filters (cuckoo or quotient filters) or larger data-base-like systems (maybe even looking at distributed scenarios). Last, one direction for concurrent hash tables we have not looked at would be space minimization.

ACKNOWLEDGMENTS

We would like to thank Markus Armbruster, Ingo Müller, and Julian Shun for fruitful discussions.

REFERENCES

[1] Lada A. Adamic and Bernardo A. Huberman. 2002. Zipf's law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.
[2] Robert L. Axtell. 2001. Zipf distribution of U.S. firm sizes. *Science* 293, 5536 (2001), 1818–1820.
[3] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Domink Schultes. 2007. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments (ALENEX'07)*. 46–59.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the18th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM'99)*. Vol. 1, 126–134. DOI: https://doi.org/10.1109/INFCOM.1999.749260

[5] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans Database Syst.* 32, 3 (2007), 17.

[6] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. 2004. Engineering a sorted list data structure for 32 bit keys. In *Proceedings of the 6th Workshop on Algorithm Engineering & Experiments (ALENEX'04)*. 142–151.

[7] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A reliable randomized algorithm for the closest-pair problem. *J. Algor.* 25, 1 (1997), 19–51.

[8] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoret. Comput. Sci.* 380, 1–2 (2007), 47–68.

[9] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. 2005. Lock-free dynamic hash tables with open addressing. *Distrib. Comput.* 18, 1 (2005). DOI: https://doi.org/10.1007/s00446-004-0115-2

[10] Torben Hagerup and Christine Rüb. 1989. Optimal merging and sorting on the EREW-PRAM. *Inform. Process. Lett.* 33 (1989), 181–185.

[11] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming*. Elsevier.

[12] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Distributed Computing*. Springer, 350–364.

[13] Euihyeok Kim and Min-Soo Kim. 2013. Performance analysis of cache-conscious hashing techniques for multi-core CPUs. *Int. J. Control Autom.* 6, 2 (2013).

[14] Donald E. Knuth. 1998. *The Art of Computer Programming—Sorting and Searching* (2nd ed.). Vol. 3. Addison Wesley.

[15] Doug Lea. 2003. Hash table util. concurrent. ConcurrentHashMap, revision 1.3. JSR-166, the Proposed Java Concurrency Package. Retrieved from http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent.

[16] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, Article 27.

[17] Tobias Maier. 2018. GrowT. Retrieved from https://github.com/TooBiased/growt.

[18] Tobias Maier and Peter Sanders. 2017. Dynamic space efficient hashing. In *Proceedings of the European Symposium on Algorithms (ESA'17)*, Vol. 87. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[19] Tobias Maier, Peter Sanders, and Roman Dementiev. 2016. Concurrent hash tables: Fast and general?(!). *CoRR* abs/1601.04017 (2016). Retrieved from http://arxiv.org/abs/1601.04017.

[20] Tobias Maier, Peter Sanders, and Roman Dementiev. 2016. Concurrent hash tables: Fast and general(?)!. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. Article 34. DOI: https://doi.org/10.1145/2851141.2851188

[21] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8 (1998), 3–30. Retrieved from http://www.math.keio.ac.jp/ matumoto/emt.html.

[22] Edward M. McCreight. 1976. A space-economical suffix tree construction algorithm. *J. ACM* 23, 2 (Apr. 1976), 262–272.

[23] Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. *Parallel Distrib. Comput. Syst.* (1998), 509–518.

[24] Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures—The Basic Toolbox*. Springer.

[25] Scott Meyers. 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Pearson Education.

[26] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1123–1136.

[27] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab et al. 2013. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 13. 385–398.

[28] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan Mcelroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani, and Facebook Inc. 2018. folly version 57:0. Retrieved from https://github.com/facebook/folly.

[29] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan Mcelroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani, and Facebook Inc. 2013. Scaling memcached at facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*.

[30] Philippe Oechslin. 2003. Making a faster cryptanalytic time-memory trade-off. In *Proceeeings of the 23rd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'03)*. Springer.

[31] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. 1995. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 175–186. DOI: https://doi.org/10.1145/223784.223813

[32] Mathieu Desnoyers Paul E. McKenney and Lai Jiangshan. 2013. LWN: URCU-protected hash tables. Retrieved from http://lwn.net/Articles/573431/.

[33] Chuck Pheatt. 2008. Intel; threading building blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298–298.

[34] Jeff Preshing. 2016. Junction. Retrieved from https://github.com/preshing/junction.

[35] Jeff Preshing. 2016. New Concurrent Hash Maps for C++. Retrieved from http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/.

[36] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *J. ACM* 53, 3 (May 2006), 379–405. DOI: https://doi.org/10.1145/1147954.1147958

[37] Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[38] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: The problem based benchmark suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 68–70.

[39] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. 2010. Lock-free parallel dynamic programming. *J. Parallel and Distrib. Comput.* 70, 8 (2010).

[40] Tony Stornetta and Forrest Brewer. 1996. Implementation of an efficient parallel BDD package. In *Proceedings of the 33rd Design Automation Conference*. ACM, 641–644.

Received September 2016; revised April 2018; accepted December 2018